PATENT ABSTRACTS OF JAPAN

(11)Publication number : 10-133884
(43)Date of publication of application : 22.05.1998

(51)Int.Cl. G06F 9/45

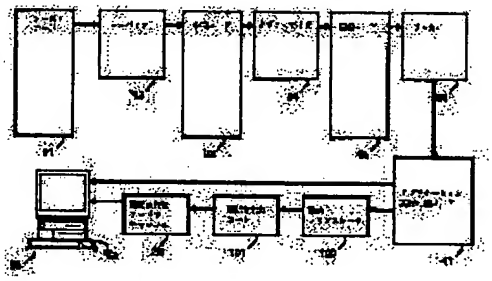
(21)Application number : 09-272305
(22)Date of filing : 06.10.1997

(71)Applicant : HEWLETT PACKARD CO <HP>
(72)Inventor : WILLIAM B BUSBY

(30)Priority
Priority number : 96 726760 Priority date : 10.10.1996 Priority country : US

(54) METHOD FOR EXECUTING PROGRAMMING CODE INCLUDING CONJECTURAL CODE

(57)Abstract:
PROBLEM TO BE SOLVED: To more efficiently utilize memory space by generating a restoration code for restoring a memory fault at run time.
SOLUTION: At the time of using a dynamic translator 100, a code block from an application executable code 41 is translated at the time of execution (run time). An optimizer 94 does not generate a conjectural restoration code, but the translator 100 generates all required conjectural restoration codes at the run time. The conjectural restoration code becomes a part of the translated code 101 and is stored n a translated code cache 20. In order to generate a conjectural restoration code, the translator 100 substantially generates a translated code for reexecuting all operations executed by using a 'dust' value generated by referring to a conjectural memory prior to the detection of a memory fault.



LEGAL STATUS

[Date of request for examination] 07.06.2004
[Date of sending the examiner's decision of rejection]
[Kind of final disposal of application other than the examiner's decision of rejection or application converted registration]
[Date of final disposal for application]
[Patent number]
[Date of registration]
[Number of appeal against examiner's decision of rejection]

BEST AVAILABLE COPY

(19)日本国特許庁（ＪＰ）　(12)**公開特許公報**（Ａ）　(11)特許出願公開番号

特開平10－133884

(43)公開日　平成10年(1998)５月22日

(51)Int.Cl.[6]	識別記号	ＦＩ	
Ｇ０６Ｆ　9/45		Ｇ０６Ｆ　9/44	３２２Ｆ

審査請求　未請求　請求項の数１　ＯＬ　（全９頁）

(21)出願番号　特願平9－272305

(22)出願日　平成９年(1997)10月６日

(31)優先権主張番号　７２６，７６０

(32)優先日　1996年10月10日

(33)優先権主張国　米国（ＵＳ）

(71)出願人　590000400
ヒューレット・パッカード・カンパニー
アメリカ合衆国カリフォルニア州パロアルト　ハノーバー・ストリート　3000

(72)発明者　ウィリアム・ピー・パズビー
アメリカ合衆国94019カリフォルニア州ハーフ・ムーン・ベイ、カサ・デル・マー・ドライブ　404

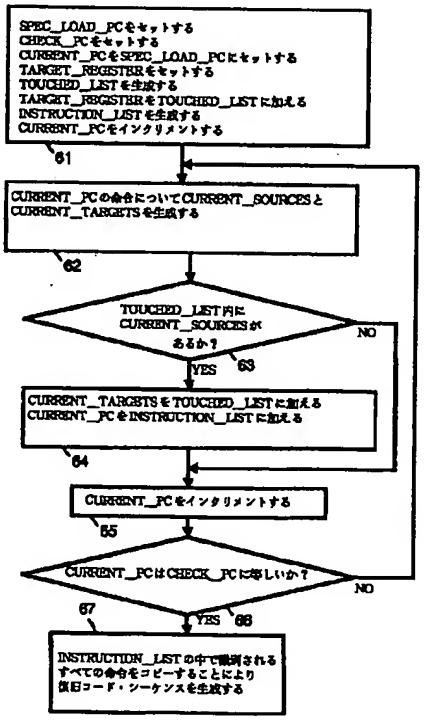
(74)代理人　弁理士　岡田　次生

(54)【発明の名称】　推測的なコードを含むプログラミング・コードを実行する方法

(57)【要約】

【課題】　推測的なコードを実行するときにメモリ・フォルトが生じる場合、メモリ空間を効率的に利用しながら、メモリ・フォルトを復旧させるコードを生成する。

【解決手段】推測的なコード・シーケンスが実行されるとき、メモリ・アクセスを含む命令についてメモリ・フォルトは遅延される。推測的なコード・シーケンスの間に生成されたデータを利用するとき、推測的なコード・シーケンスが実行されたときにメモリ・フォルトが生じたかどうか判断する。推測的なコード・シーケンスが実行されたときにメモリ・フォルトが生じたと判断される場合、例えば動的トランスレータが復旧コードを生成する。復旧コードは、それが実行されるとき、メモリ・フォルトによって改悪されたあらゆる演算を再び実施する。



1

【特許請求の範囲】

【請求項１】推測的に実行される推測的なコードを含むプログラミング・コードを実行する方法であって、

推測的なコード・シーケンスを実行するステップであって、推測的なコード・シーケンス内のメモリ・アクセスを含む命令についてメモリ・フォルトを無視することを含むステップと、

上記推測的なコード・シーケンスの間に生成されるデータを利用するとき、上記推測的なコード・シーケンスが実行されたときにメモリ・フォルトが生じたかどうか判断するステップと、

上記判断ステップで、上記推測的なコード・シーケンスが実行されたときにメモリ・フォルトが生じたと判断される場合、実行時に上記メモリ・フォルトからの復旧を実施する復旧コードを生成するステップと、を含む上記方法。

【発明の詳細な説明】

【０００１】

【発明の属する技術分野】この発明は、コンピュータ・システム上のプログラムの実行に関し、特に、推測の復旧コードを生成するための、ランタイムのコード生成の使用に関する。

【０００２】

【従来の技術】プログラムは、一般に高水準プログラミング言語で書かれる。この高水準言語は、しばしばソース・コードと呼ばれ、コンパイラ・プログラムによってアセンブリ言語に翻訳される。アセンブリ言語のバイナリ形式は、目的コードと呼ばれ、コンピュータによって実際に実行されるコード形式である。一般に目的コードは、最初にリンカーによって互いに連係される目的コード・モジュールに作られる。ここで、「コンパイル」という用語は、目的コード・モジュールを作り、その目的コード・モジュールを互いに連係させるプロセスを含む。

【０００３】最新の最適化技術の１つは、推測的(speculative)な実行、または推測(speculation)である。推測的な実行を実施するために、コンパイラは、将来必要になると思われる動作が、早めにまたは推測的に実施されるように、コードを配列する。早めに実行されるコードの一部分は、推測的なコードと呼ばれる。推測的なコードによって実施される動作が行われる必要があると判明した場合、推測的なコードは早めに実行されているので、これは、コードの実行の速度を上げる。その動作が必要でないと判明する場合、推測的なコードによって計算された結果は捨てられる。

【０００４】

【発明が解決しようとする課題】推測に必要な条件は、推測的に実行されるコードが、推測されないバージョンと異なって、外部から見える動作を引き起こさないことである。しかし、推測の使用は、ある動作の実行の相対

2

時間を再配置するので、例えばメモリ・フォルト(memory fault)を処理するためにトラップ・ハンドラの起動を引き起こす推測的なコードの例外は、しばしば適正な演算のために特別な復旧コードの実行を必要とする。そのような場合、非常に積極的な推測の使用は、莫大な量の復旧コードを必要とすることがある。しかしこの復旧コードは、実行可能ファイル内に空間をとる。任意の特定の復旧コードが利用される必要は滅多にないので、これは、メモリ空間の非効率的な使用である。

【０００５】

【課題を解決するための手段】この発明の好ましい実施例に従って、プログラミング・コードは、推測的なコードを含む。推測的なコードは、推測的なコードの実行に起因する結果が将来必要とされるという推測に基づいて、早めに実行されるコードである。推測的なコード・シーケンス内のメモリ・アクセスを含む命令を実行しているとき、いかなるメモリ・フォルトも無視される。推測的なコード・シーケンスの間に生成されたデータを利用する際、推測的なコード・シーケンスが実行されたときにメモリ・フォルトが生じたかどうかについて判断がなされる。推測的なコード・シーケンスが実行されたときにメモリ・フォルトが生じたと判断される場合、復旧コードが生成され、それは実行されるとき、メモリ・フォルトを復旧させる。

【０００６】例えば、動的トランスレータが復旧コードを生成し、その復旧コードは、実行されるとき、メモリ・フォルトによって改悪された演算を再び実施する。この発明のある実施例で、動的トランスレータは、最適化コンパイラによって提供されるコード注釈を使用する。最適化コンパイラは、コンパイル時間中、コード注釈をプログラミング・コードに挿入する。この発明の別の実施例では、動的トランスレータは、最適化コンパイラによって提供される復旧コードのコンパクトな表現を使用する。最適化コンパイラは、コンパイル時間中、復旧コードのコンパクトな表現をプログラミング・コードに挿入する。

【０００７】この発明は、メモリ・フォルトを復旧させる復旧コードのランタイム生成を提供することにより、プログラミング・コード内の空間のより効率的な利用を可能にする。メモリ・フォルトは、推測的に実行されるコード内で生じ、推測的に実行されるコードの結果の使用中に検知される。

【０００８】

【発明の実施の形態】図１は、あるアプリケーションのための実行可能コードを作るために、コンパイラ・システムを使用するコンピュータ・システムのブロック図を示す。コンパイラ92は、ソース・コード91を受け取り、中間コード93を作る。中間コードは、目的(アセンブリ)言語命令のリストである。オプティマイザ94は、中間コード93を受け取り、最適化された目的コード95を作る。

リンカー96は、最適化された目的コード95を受け取り、アプリケーション実行可能コード41(アプリケーション41とも呼ばれる)を作る。アプリケーション実行可能コード41は、コンピューティング・システム98によって実行することができる。コンパイラ92、オプティマイザ94およびリンカー96は、一緒に最適化コンパイラを形成する。コンパイラ92、オプティマイザ94および/またはリンカー96によって実施される動作は、コンパイル時間中に行われる。

【０００９】図１に示されるコンピュータ・システムは、翻訳されたコードをランタイム中に作り出すために使用される動的トランスレータ100を含む。動的トランスレータ100を使用するとき、アプリケーション実行可能コード41からのコード・ブロックは、実行時(ランタイム)に翻訳される。翻訳されたコード・ブロックは、翻訳されたコード101として図１に示される。翻訳されたコード101は、メモリ(例えばキャッシュメモリ)に格納されるので、何回か実行されるそれぞれのコード・ブロックは、ランタイム中に一度だけ翻訳されればよい。このアプローチは、コードをランタイム前に翻訳する必要がないという柔軟性を与えるにもかかわらず、コード・ブロックが実行される度に翻訳される場合の経費を減少させる。

【００１０】好ましい実施例で、翻訳されたコード101は、翻訳されたコードのキャッシュ20に格納される。翻訳されたコードのキャッシュ20が満杯であるとき、新しく翻訳されたコード・ブロックのための場所を作るために、以前に翻訳されたコード・ブロックを捨てなければならないことがある。これは、捨てられたコード・ブロックが再び使用される場合、再び翻訳されることを必要とするが、メモリ使用の可能な節減を与える。代わりとして、以前に翻訳されたコード・ブロックを捨てるのではなく、それらをシステム・メモリに格納することもできる。

【００１１】翻訳は、推測的に実行されるコードの正しい実行に必要とされる復旧コードを生成するために使用＊

＊することができる。具体的に言うと、推測的な実行を実施するために、コンパイラは、将来必要になると思われる動作がより早めにまたは推測的に実施されるようにコードを配列する。その動作が行われる必要があると判明した場合、それらはすでに行われているので、これはコードの実行の速度を上げる。その動作が必要でないことが判明した場合、推測的に計算された結果は捨てられる。

【００１２】例えば下の表１は、コードの一例を示す。その例は、プログラミング言語Ｃで示される。

【００１３】

【表１】

```
if((a/b)>3.2) {
        c = *d+1;
}
```

【００１４】表１に示されるコードの例で、変数「a」および「b」は浮動小数点数であり、変数cは整数であり、変数「*d」はポインタ「d」によって指し示される整数である。ステートメント「c = *d+1」は、条件節((a/b)>3.2)が真である場合に実行される必要がある。通常コンパイラ92は、条件節((a/b)>3.2)が計算され、その条件節((a/b)>3.2)が真である場合のみ次のステートメント「c = *d+1」が実行されるようにコードを生成する。

【００１５】しかし、多くの最新のコンピュータは、並行に演算することができる複数の算術機能ユニットを持つ。そのようなコンピュータで、ステートメント「c = *d+1」の部分を条件節((a/b)>3.2)より先にまたは並行に実行すると、上記の表１で示されるようなコードの実行の速度を上げることができる。そのような並行な実行を容易にするために、オプティマイザ94は、表１のコードの例を、以下の表２のコードのように並べ替える。

【００１６】

【表２】

```
temp = *d + 1;                    [推測的に実行されるコード]
if((a/b)>3.2) {
        c = temp;
}
```

【００１７】表２に示すコードの例で、ステートメント「c = temp」は、条件節((a/b)>3.2)が真である場合のみ実行される。条件節((a/b)>3.2)が偽である場合、「temp」に格納された、推測的に実行された結果(*d+1)は、使用されないだけであり、効果的に捨てられる。

【００１８】表２の単なるコードの並べ替えは、問題を提起することがある。具体的にいうと、推測に必要な条件は、推測的に実行されるコードが、推測されないバージョンと異なって、外部から見える行動を引き起こさないことである。表２に示されるコードで、ポインタ「d」が無効である場合、無効な参照を行う(de-referencing)ポインタ「d」(すなわち整数「*d」にアクセスするためにポインタ「d」を使用する)は、メモリ・フォルトを引き起こす。このメモリ・フォルトは、表２のコードが実行されるときに生じる。

【００１９】しかし表１のコードを実行するとき、条件節((a/b)>3.2)が偽である場合は必ず、ポインタ「d」が、無効な参照をされることはなく、メモリ・フォルトは起こらない。表２のコードを実行することは、結果として表１のコードを実行するときに検知されないメモリ

5

・フォルトを処理することになりうるので、表２の推測的なコードを実行することは、表１のコードの推測されないバージョンとは異なって、外部から見える動作を生じさせることがある。

【００２０】上記の問題の解決策は、推測的に実行されるコードが本当に必要であることが確実になるまで、推測的に実行されるメモリ参照のときのフォルトを遅らせるメカニズムを構造上定義することである。推測的に実行されるコードが本当に必要であるとき、メモリ・フォルトを不適切に表すということを心配せずにメモリ参照＊

6

＊を行うことができる。これは、以下の表３および表４に示されるアセンブリ言語コードで説明される。表３は、上記の表１に示されたコードのアセンブリ言語バージョンを示す。表３のコードは、プレシジョン・アーキテクチャ(PrecisionArchitecture、PA)1.1のために使用されるアセンブリ言語である。プレシジョン・アーキテクチャに関する更なる情報は、ヒューレット・パッカード社から得ることができる。

【００２１】

【表３】

```
                //**if((a/b)>3.2){**//
    FDIV a,b,ftemp          ;aをbで割る
    FCMP,>ftemp,3,2         ;3.2と比較する
    B,n out                 ;節に基づく条件付き分岐
            //**c = *d + 1**//
    LDW 0(d),temp           ;dの無効な参照を行う(de-reference)
    ADDI 1,temp,temp        ;*dに1を加える
    COPY temp,c             ;インクリメントされた値をcに入れる
out:
```

【００２２】表４は、表３に示されるたアセンブリ言語の推測的なアセンブリコード・バージョンを示す。表４に示される推測的なアセンブリコード・バージョンの中で、推測的に実行されるメモリ参照のときのフォルトは、推測的に実行されるコードが本当に必要であること※

※が確実になるまで遅らされる。推測的に実行されるコードが必要であると判断されるとき、遅延されたメモリ・フォルトが提示される。

【００２３】

【表４】

```
                //**temp = *d + 1**//
    LDW,speculative 0(d),temp  ;トラッピングを遅延させるLDWの
                               ;推測的バージョンを使用して、
                               ;dの無効な参照を行う。
    ADDI 1,temp,temp           ;*dに1を加える
            //**if((a/b)>3.2){**//
    FDIV a,b,ftemp             ;aをbで割る
    FCMP,> ftemp,3,2           ;3.2と比較する
    B,n out                    ;節に基づく条件付き分岐
            //**c = temp**//
    CHECK 0(d),temp            ;dの有効性を調べ、トラップする必要が
                               ;あるか確認し、
                               ;もしそうである場合、トラップする。
    COPY temp,c                ;インクリメントされた値をcに入れる
out:
```

【００２４】表４のコードは、推測的なコードを説明する。推測的なコード内のコードがメモリ参照を行うとき、メモリ・フォルトは、推測的なコードが実行されたと推測を含まないコードの中で判断されるまでは引き起こされない。

【００２５】表４のコードには、２つの新しい命令「LDW,speculative」および「CHECK」がある。「LDW,speculative」は、それがいかなるメモリ・フォルトも起こさないという点を除いて、表３のコードで使用されるLDWと機能が類似している。さらに、「LDW,speculative」を実行するときにメモリ・フォルトが生じていた場合、目標レジスタにロードされる値は不確定である。

【００２６】命令「CHECK」は、LDWが実行されたときメモリ・フォルトが生じたかどうかを単純に確かめることを除いて、表３のコードの中で使用されるLDWと機能が類似している。LDWが実行されたときにメモリ・フォルトが生じた場合、CHECKは、通常はLDWの実行中に生じるトラッピングを実施する。

【００２７】多くの最新のコンピュータ・アーキテクチャで、プログラムは、フォルトを引き起こした問題を解決することによって、メモリ・フォルトから復旧することができる。これは、メモリ参照命令をやり直し、実行

を続けることによって実施される。上述したように、これは、推測がメモリ・フォルトの処理に遅れを生じさせたときに本当の問題を提示する。これは、メモリ・フォルトを検知しない結果として、無効なポインタでメモリにアクセスする推測的な命令を実行するときに、「ごみ」値("garbage" value)をロードすることがあるからである。例えば、上記の表４で与えられるコードについて、ポインタ「d」には無効なポインタが入っていると仮定する。もしそうであれば、ポインタ「d」を使用し＊

```
                    //**temp = *d+1**//
        LDW,speculative 0(d),temp   ;トラッピングを遅延させるLDWの
                                    ;推測的バージョンを使用して、
                                    ;dの無効な参照を行う(de-reference)。
```

##ポインタ「d」は無効であるので、通常、LDWコマンド※はメモリ・フォルトを引き起こす。しかしLDWの推測的※

```
        ADDI 1,temp,temp            ;*dに1を加算する
```

##値「*d」は、インクリメントされる。しかし、ポインタ「d」は無効であり、*dはごみ値であるので、インク★

```
                    //**if((a/b)>3.2){**//
        FDIV a,b,ftemp              ;aをbで割る
        FCMP,> ftemp,3,2            ;3.2と比較する
        B,n out                     ;節に基づく条件付き分岐
                    //**c = temp**//
        CHECK 0(d), temp            ;dの有効性を調べ、トラップが必要であるか
                                    ;どうか確認し、
                                    ;そうである場合、トラップする。
```

##CHECK命令で、ポインタ「d」が悪い値であると認識され、トラップが起こる。トラップ・ハンドラは、ポインタ「d」を良いポインタに訂正し、演算を再び行うことを要求する。推測が行われなかったならば、トラップ・ハンドラが訂正を行ってから、値「*d」を値「temp」に☆

```
        COPY temp,c                 ;インクリメントされた値をcに入れる
    out:
```

【００２９】推測されるコード内のメモリ・フォルトから如何にして復旧するかという、表５のコードの注釈に記述される問題の１つの解決策は、オプティマイザ94で推測の復旧コードを生成することであり、その推測の復旧コードは、推測的に実行されるコード内でメモリ・フォルトが生じるときに利用することができる。

【００３０】例えば、表５に示されるコードについて推測的な復旧コードは、下の表６に示されるコードの列(line)である。

【００３１】

【表６】ADDI 1,temp,temp

【００３２】演算「CHECK」が実施されるとき、メモリ・フォルトの発見より先に、推測的なメモリ参照によって生成される「ごみ」値を使用して行われたあらゆる演算を再び実施する。

【００３３】この解決策に伴う１つの問題は、復旧コードは空間に関して高価になりうるということである。オ

＊て参照されるデータは、ごみ値になる。このごみ値が他の命令によって使用される場合、これらの他の演算の結果もまた改悪される。メモリ・フォルトが見つかったとき、ごみ値によって汚染された、改悪された値を再計算することが必要である。下の表５の中の注釈は、無効なポインタが推測的なコードの中で使用されるときに生じる事象を示す。

【００２８】

【表５】

※コマンドが使用されるので、メモリ・フォルトおよび関連するトラップは遅延される。

★リメントの結果もまたごみ値である。

☆ロードすることができる。しかし、推測が生じ、すでにインクリメントが行われ、何らかの復旧が行われていないこのケースでは、cにロードされる値にインクリメントが反映されず、それゆえコードは、適切に演算しない。

プティマイザ94は、多くの推測的な演算を利用することを要求するが、復旧コードは、実行可能ファイル内に相当量の空間を取りうる。一般にプログラムは、上述されたある種のメモリ・フォルトの復旧を実施する必要が滅多にないので、これは空間の非効率的な使用を表す。

【００３４】この発明の好ましい実施例で、オプティマイザ94は、推測的な復旧コードを生成しない。むしろ動的トランスレータ100が、必要とされるあらゆる推測的な復旧コードをランタイムに生成する。この推測的な復旧コードは、翻訳されたコード101の一部となり、翻訳されたコードのキャッシュ20に格納することができる。

【００３５】推測的な復旧コードを生成するため、本質的に動的トランスレータ100は、メモリ・フォルトの発見より先に、推測的なメモリ参照により生成された「ごみ」値を使用して実行されたあらゆる演算を再び実施する翻訳されたコードを生成する。

【００３６】図２は、動的トランスレータ100が、どの

ようにしてアプリケーション実行可能コード41内の目的コードを解析し、復旧コード・シーケンスを生成するかを示すフローチャートである。フローチャートは、ユーザが、遅延されるトラップから復旧したいと望むときに適用する。

【００３７】ステップ61で、多様な初期化が実施される。変数SPEC_LOAD_PCは、遅延されるトラップに関連する、推測されるメモリ参照命令のアドレスにセットされる。変数CHECK_PCは、遅延されるトラップを引き起こした命令のアドレスにセットされる。変数CURRENT_PCは、変数SPEC_LOAD_PCの現在値にセットされる。変数TARGET_REGISTERは、SPEC_LOAD_PCにおける命令の目標レジスタにセットされる。TOUCHED_LISTは、レジスタのNULLリストとして生成され、初期化される。そしてTARGET_REGISTERは、TOUCHED_LISTに加えられる。INSTRUCTION_LISTは、命令アドレスのNULLリストとして生成され、初期化される。それからCURRENT_PCは、インクリメントされてCURRENT_PCの後にくる命令を指示する。

【００３８】ステップ62で、CURRENT_PCにおける命令が調べられ、そのソース・レジスタ(CURRENT_SOURCES)全部およびその目標レジスタ(CURRENT_TARGETS)全部のリストが生成される。

＊【００３９】ステップ63で、CURRENT_SOURCESのどれかがTOUCHED_LIST内に含まれるかどうかについて判断がなされる。そうである場合、ステップ64で、CURRENT_TARGETSはTOUCHED_LISTに加えられ、CURRENT_PCはINSTRUCTION_LISTに加えられる。ステップ63で、CURRENT_SOURCESのどれもTOUCHED_LIST内に含まれない場合、ステップ64はスキップされる。

【００４０】ステップ65で、CURRENT_PCは、インクリメントされてCURRENT_PCのに後に続く命令を指示する。ステップ66で、CURRENT_PCがCHECK_PCと等しいかどうかに関して判断がなされる。そうでない場合、ステップ62から66までが繰り返される。ステップ66で、CURRENT_PCがCHECK_PCに等しい場合、ステップ67で、復旧コード・シーケンスが、INSTRUCTION_LISTの中に識別される全ての命令をコピーすることによって生成される。

【００４１】下の表７は、図２に示されるフローチャートを実現する擬似コードを示し、さらに、動的トランスレータ100がどのようにしてアプリケーション実行可能コード41内の目的コードを解析し、復旧コード・シーケンスを生成するかを示す。

【００４２】

＊　【表７】

```
if(ユーザが、遅延されるトラップから復旧することを望む場合){
    (SPEC_LOAD_PCを、遅延されるトラップに関連する、推測されるメモリ参照命令のアドレスにセットする)
    (CHECK_PCを、遅延されるトラップを引き起こした命令のアドレスにセットする)
    (CURRENT_PCをSPEC_LOAD_PCにセットする)
    (SPEC_LOAD_PCにおける命令を調べ、TARGET_REGISTERをその目標レジスタにセットする)
    (TOUCHED_LISTと命名されるレジスタのNULLリストを生成する)
    (TARGET_REGISTERをTOUCHED_LISTに加える)
    (INSTRUCTION_LISTと命名される命令アドレスのNULLリストを生成する)
    (CURRENT_PCを、CURRENT_PCに続く命令にセットする)
    (DONE=FALSEにセットする)
    while (DONEが偽である間) do {
        (CURRENT_PCにおける命令を調べ、そのソース・レジスタ(CURRENT_SOURCES)全部およびその目標レジスタ(CURRENT_TARGETS)全部のリストを生成する)
        if(CURRENT_SOURCESのどれかがTOUCHED_LIST内に含まれる場合) then
            (CURRENT_TARGETSをTOUCHED_LISTに加え,CURRENT_PCをINSTRUCTION_LISTに加える)
            (CURRENT_PCを、CURRENT_PCに続く命令にセットする)
        if(CURRENT_PCがCHECK_PCと等しい)  then (DONEを真にセットする)
    }
    (INSTRUCTION_LIST内で識別される全ての命令をコピーすることにより復旧コード・シーケンスを生成する)
}
```

【００４３】ある場合に、特に制御フローが、CURRENT_PCの後にどの命令が続くかをはっきりさせないとき、補足の情報が、例えばオプティマイザ94によってコード注釈の形で供給される。制御フローを解析するとき、これらのコード注釈は、動的トランスレータ100によって利用される。

【００４４】この発明の別の代替の実施例で、オプティマイザ94は、必要な復旧コードのコンパクトな表現を生成する。この復旧コードのコンパクトな表現は、それが必要とされるときに翻訳され、あるいは復旧コード生成のためのテンプレートとして使用される。この実施例は、復旧コードを生成する動的トランスレータ100の使用に関連して用いることができる。

【００４５】本発明は例として次の実施態様を含む。

【００４６】（１）推測的に実行される推測的なコードを含むプログラミング・コードを実行する方法であって、

(a) 以下のサブステップ(a.1)を含む、推測的なコード・シーケンスを実行するステップと、

(a.1) メモリ・アクセスを含む推測的なコード・シーケンスの中の命令について、メモリ・フォルトを無視するステップ、

(b) 推測的なコード・シーケンスの間に生成されるデータを利用するとき、以下のサブステップ(b.1)、(b.2)を実行するステップと、

(b.1) 推測的なコード・シーケンスが実行されたときにメモリ・フォルトが起こったかどうか判断するステップ、

(b.2) サブステップ(b.1)で、推測的なコード・シーケンスが実行されたときにメモリ・フォルトが起きたと判断されるとき、以下のサブステップ(b.2.1)を実施するステップ、(b.2.1) 実行されるときにメモリ・フォルトからの復旧を実施する復旧コードを生成するステップ、を含む上記方法。

【００４７】（２）サブステップ(b.2.1)は、実行されるとき、メモリ・フォルトによって改悪されたあらゆる演算を再び実施する翻訳されたコードを生成することを含む、上記(1)の方法。

（３）サブステップ(b.2)はさらに、(b.2.2) サブステップ(b.2.1)で生成された復旧コードを実行するステップ、を含む、上記(1)の方法。

【００４８】（４）サブステップ(b.2.1)は、最適化コンパイラによって提供されるコード注釈を使用することを含み、上記最適化コンパイラは、コンパイル時間中にコード注釈をプログラミング・コードへ挿入する、上記(1)の方法。

（５）サブステップ(b.2.1)は、最適化コンパイラによって提供される復旧コードのコンパクトな表現を使用することを含み、上記最適化コンパイラは、コンパイル時間中に、復旧コードのコンパクトな表現をプログラミング・コードへ挿入する、上記(1)の方法。

（６）サブステップ(b.2.1)は、動的トランスレータによって実施される、上記(1)の方法。

【００４９】（７）実行されるとき、推測的に実行される推測的なコードを含むプログラミング・コードを実行する方法を実施するソフトウェアを格納する記憶媒体であって、

(a) 推測的なコード・シーケンスを実行するステップであって、

(a.1) メモリ・アクセスを含む推測的なコード・シーケンスの中の命令についてメモリ・フォルトを無視するサブステップ、を含むステップと、

(b) 推測的なコード・シーケンスの間に生成されるデータを利用するとき、

(b.1) 推測的なコード・シーケンスが実行されたときにメモリ・フォルトが起こったかどうか判断するサブステップと、

(b.2) サブステップ(b.1)で、推測的なコード・シーケンスが実行されたときにメモリ・フォルトが起こったと判断されるとき、(b.2.1) 実行時にメモリ・フォルトからの復旧を実施する復旧コードを生成するサブステップ、を実施するサブステップと、を実施するステップと、を含む、上記記憶媒体。

【００５０】（８）サブステップ(b.2.1)が、実行されるとき、メモリ・フォルトによって改悪されたあらゆる演算を再び実施する翻訳されたコードを生成することを含む、上記(7)の記憶媒体。

（９）サブステップ(b.2)はさらに、(b.2.2) サブステップ(b.2.1)で生成された復旧コードを実行するサブステップ、を含む上記(7)の記憶媒体。

【００５１】（１０）サブステップ(b.2.1)は、最適化コンパイラによって提供されるコード注釈を使用することを含み、上記最適化コンパイラは、コンパイル時間中にコード注釈をプログラミング・コードへ挿入する、上記(7)の記憶媒体。

（１１）サブステップ(b.2.1)は、最適化コンパイラによって提供される復旧コードのコンパクトな表現を使用することを含み、上記最適化コンパイラは、コンパイル時間中に復旧コードのコンパクトな表現をプログラミング・コードへ挿入する、上記(7)の記憶媒体。

（１２）サブステップ(b.2.1)は、動的トランスレータによって実施される、上記(7)の記憶媒体。

【００５２】（１３）推測的に実行される推測的なコードを含むプログラミング・コードと、上記プログラミング・コードを実行する実行手段であって、上記プログラミング・コードは、推測的なコード・シーケンス内のメモリ・アクセスを含む命令についてメモリ・フォルトが無視されるように最適化される手段と、メモリ・フォルトのための復旧コードを生成する動的トランスレータであって、上記メモリ・フォルトは、推測的なコード・シーケンスが実行されるときに生じ、推測的なコード・シーケンスの実行中に生成されるデータを利用するときに発見される、動的トランスレータと、を含み、上記復旧コードは、実行手段によって実行されるときにメモリ・フォルトからの復旧を実施する、コンピューティング・システム。

13

【００５３】（１４）上記動的トランスレータは、実行手段によって実行されるとき、メモリ・フォルトによって改悪されたあらゆる演算を再び実施する翻訳されたコードを生成する、上記(13)のコンピューティング・システム。

（１５）上記動的トランスレータは、上記復旧コードを生成するために、上記プログラミング・コード内に提供されるコード注釈を使用する、上記(13)のコンピューティング・システム。

（１６）上記動的トランスレータは、上記復旧コードを生成するために、上記プログラミング・コード内に提供される復旧コードのコンパクトな表現を使用する、上記(13)のコンピューティング・システム。

【００５４】（１７）さらに、上記プログラミング・コードを生成する最適化コンパイラを含む、上記(13)のコンピューティング・システム。

（１８）上記最適化コンパイラは、コンパイラおよびオプティマイザを含み、上記オプティマイザは上記推測的なコードを生成する、上記(17)のコンピューティング・システム。

【００５５】

【発明の効果】本発明の方法によれば、メモリ・フォルトを復旧させる復旧コードをランタイムに生成することによって、メモリ空間をより効率的に利用することができる。

14

【図面の簡単な説明】

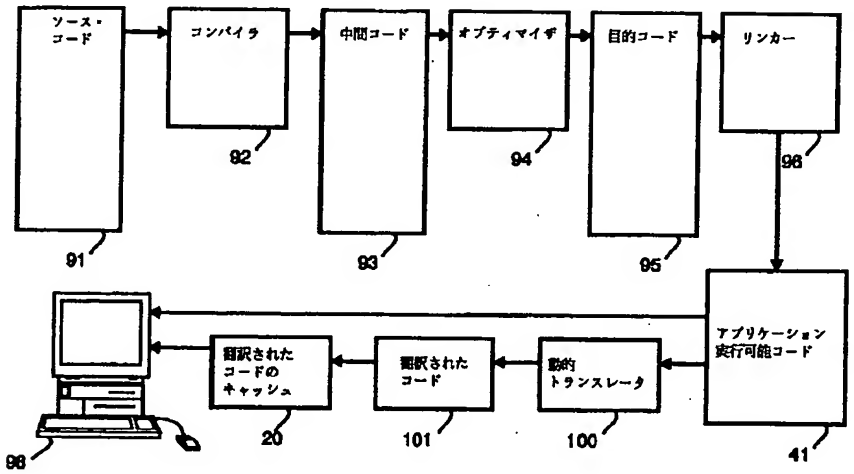
【図１】この発明の好ましい実施例に従って、コンパイラを使用して、動的トランスレータを含むシステム内で実行される実行可能コードを作るコンピュータ・システムのブロック図。

【図２】この発明の好ましい実施例に従って、動的トランスレータが、どのようにしてアプリケーション実行可能コード内の目的コードを解析し、復旧コード・シーケンスを生成するかを示すフローチャート。

【符号の説明】

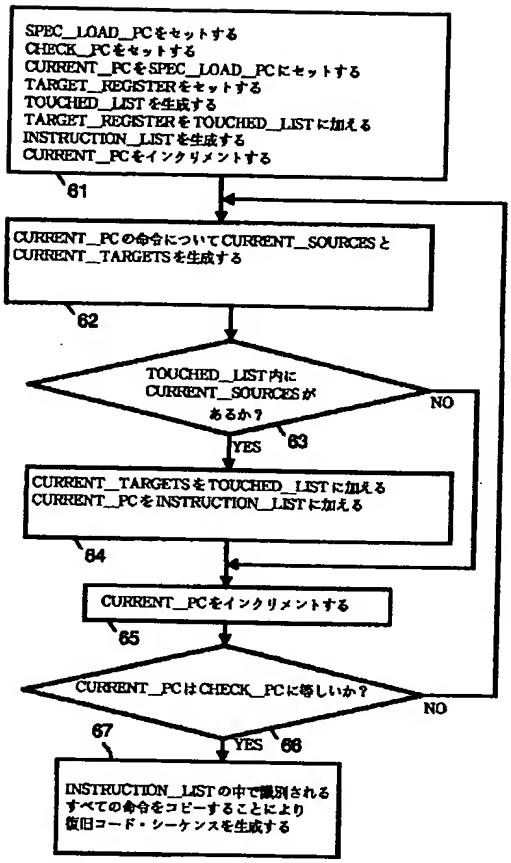
９１	ソース・コード
９２	コンパイラ
９３	中間コード
９４	オプティマイザ
９５	目的コード
９６	リンカー
４１	アプリケーション実行可能コード
１００	動的トランスレータ
１０１	翻訳されたコード
２０	翻訳されたコードのキャッシュ

【図１】



BEST AVAILABLE COPY

【図２】



BEST AVAILABLE COPY